Spatial: A Language and Compiler for Application Accelerators

David Koeplinger† Matthew Feldman† Raghu Prabhakar† Yaqi Zhang†
Stefan Hadjis† Ruben Fiszel‡ Tian Zhao† Luigi Nardi† Ardavan Pedram†
Christos Kozyrakis† Kunle Olukotun†

† Stanford University, USA
‡ École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

{dkoeplin,mattfel,raghup17,yaqiz,shadjis,rfiszel,tianzhao,lnardi,perdavan,kozyraki,kunle}@stanford.edu

Abstract

Industry is increasingly turning to reconfigurable architectures like FPGAs and CGRAs for improved performance and energy efficiency. Unfortunately, adoption of these architectures has been limited by their programming models. HDLs lack abstractions for productivity and are difficult to target from higher level languages. HLS tools are more productive, but offer an ad-hoc mix of software and hardware abstractions which make performance optimizations difficult.

In this work, we describe a new domain-specific language and compiler called Spatial for higher level descriptions of application accelerators. We describe Spatial's hardware-centric abstractions for both programmer productivity and design performance, and summarize the compiler passes required to support these abstractions, including pipeline scheduling, automatic memory banking, and automated design tuning driven by active machine learning. We demonstrate the language's ability to target FPGAs and CGRAs from common source code. We show that applications written in Spatial are, on average, 42% shorter and achieve a mean speedup of 2.9× over SDAccel HLS when targeting a Xilinx UltraScale+ VU9P FPGA on an Amazon EC2 F1 instance.

CCS Concepts • **Hardware** → **Hardware accelerators**; *Reconfigurable logic applications*; • **Software and its engineering** → **Data flow languages**; **Source code generation**;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI'18, June 18–22, 2018, Philadelphia, PA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192379

Keywords domain-specific languages, compilers, hardware accelerators, high-level synthesis, reconfigurable architectures, FPGAs, CGRAs

ACM Reference Format:
David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *PLDI '18: PLDI '18: ACM SIGPLAN Conference on Programming Language Design and Implementation, June 18–22, 2018, Philadelphia, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192379

1 Introduction

Recent trends in technology scaling, the availability of large amounts of data, and novel algorithmic breakthroughs have spurred accelerator architecture research. Reconfigurable architectures like field-programmable gate arrays (FPGAs) and coarse-grain reconfigurable architectures (CGRAs) have received renewed interest from academic researchers and industry practitioners alike, primarily due to their potential performance and energy efficiency benefits over conventional CPUs. FPGAs are now being used to accelerate web search in datacenters at Microsoft and Baidu [29, 34], Amazon now offers FPGA instances as part of AWS [4], and Intel has announced products like in-package Xeon-FPGA systems [18] and FPGA-accelerated storage systems [21]. Similarly, several recent research prototypes [17, 30–32, 40] and startups [6, 7] have explored various kinds of CGRAs at different granularities. Growing use of such reconfigurable architectures has made them more available to programmers now than ever before.

Reconfigurable devices are able to accelerate applications, in part, by exploiting multiple levels of nested parallelism and data locality with custom data pipelines and memory hierarchies. Unfortunately, the same features that make reconfigurable architectures efficient also make them much more complex to program. An accelerator design must account for the timing between pipelined signals and the physically limited compute

and memory resources available on the target device. It must also manage partitioning of data between local scratchpads and off-chip memory to achieve good data locality. The combination of these complexities leads to intractable accelerator design spaces [13].

These challenges have caused programmability to be a key limiting factor to widespread adoption of CGRAs and FPGAs [10, 15]. The space of CGRA programmability is fragmented with incompatible, architecture-specific programming models. The current state of the art in programming FPGAs involves using a combination of vendor-supplied IP blocks, hand-tuned hardware modules written using either low-level RTL or high-level synthesis tools, and architecture-specific glue logic to communicate with off-chip components such as DRAM. Hardware description languages (HDLs) like Verilog and VHDL are designed for explicit specification of hardware, placing the burden on the user to solve the complexities of implementing their algorithm in hardware.

High-level synthesis (HLS) tools like SDAccel [42], Vivado HLS [3], and Intel's OpenCL SDK [5] raise the level of abstraction compared to HDLs significantly. For example, HLS tools allow programmers to write accelerator designs in terms of untimed, nested loops and offer library functions for common operations like data transfer between a CPU host and the FPGA. However, existing commercial HLS tools have all been built on top of software languages like C, OpenCL, and Matlab. These software languages have been built to target instruction-based processors like CPUs and GPUs. Consequently, although existing HLS tools raise the level of abstraction for targeting reconfigurable architectures, they do so with an ad-hoc, often underspecified mix of software and hardware abstractions. For instance, while SDAccel can convert nested loops into hardware state machines, the language has no notion of the architecture's memory hierarchy and cannot pipeline loops at arbitrary nesting levels [2]. Programmers must keep in mind that, despite the software programming abstractions, they must employ hardware, not software, optimization techniques. This makes it challenging to write HLS code which produces fully optimized designs [26].

In this work, we first summarize high-level language abstractions required to create a new high-level synthesis language from the ground up, including syntax for managing memory, control, and accelerator-host interfaces on a reconfigurable architecture. We suggest that this "clean slate" approach to high-level synthesis language design leads to a language which is semantically cleaner when targeting reconfigurable architectures, particularly when optimizing for data locality and parallelism. These abstractions help programmer productivity and allow both the user and compiler to more easily optimize designs for improved performance.

```
1  // Custom floating point format
2  // 11 mantissa, 5 exponent bits
3  type Half = FltPt[11,5]
4
5  def main(args: Array[String]) {
6
7    // Load data from files
8    val a: Matrix[Half] = loadMatrix[Half](args(0))
9    val b: Matrix[Half] = loadMatrix[Half](args(1))
10
11   // Allocate space on accelerator DRAM
12   val A = DRAM[Half](a.rows,a.cols)
13   val B = DRAM[Half](b.rows,b.cols)
14   val C = DRAM[Half](a.rows,b.cols)
15
16   // Create explicit design parameters
17   val M = 128 (64, 1024)  // Tile size for output rows
18   val N = 128 (64, 1024)  // Tile size for output cols
19   val P = 128 (64, 1024)  // Tile size for common
20   val PAR_K  = 2 (1, 8)   // Unroll factor of k
21   val PAR_J  = 2 (1, 16)  // Unroll factor of j
22
23   // Transfer data to accelerator DRAM
24   sendMatrix(A, a)
25   sendMatrix(B, b)
26
27   // Specify the accelerator design
28   Accel {
29     // Produce C in M x N tiles
30     Foreach(A.rows by M, B.cols by N){ (ii,jj) =>
31       val tileC = SRAM[Half](M, N)
32
33       // Combine intermediates across common dimension
34       MemReduce(tileC)(A.cols by P){ kk =>
35         // Allocate on-chip scratchpads
36         val tileA = SRAM[Half](M, P)
37         val tileB = SRAM[Half](P, N)
38         val accum = SRAM[Half](M, N)
39
40         // Load tiles of A and B from DRAM
41         tileA load A(ii::ii+M, kk::kk+P)  // M x P
42         tileB load B(kk::kk+P, jj::jj+N)  // P x N
43
44         // Combine intermediates across a chunk of P
45         MemReduce(accum)(P by 1 par PAR_K){ k =>
46           val partC = SRAM[Half](M, N)
47           Foreach(M by 1, N by 1 par PAR_J){ (i,j) =>
48             partC(i,j) = tileA(i,k) * tileB(k,j)
49           }
50           partC
51         // Combine intermediates with element-wise add
52         }{(a,b) => a + b }
53       }{(a,b) => a + b }
54
55       // Store the tile of C to DRAM
56       C(ii::ii+M, jj::jj+N) store tileC
57     }
58   }
59
60   // Save the result to another file
61   saveMatrix(args(2), getMatrix(C))
62 }
```

Figure 1. Basic matrix multiplication ($C = A \cdot B$) implemented in Spatial.

We then describe a new domain specific language (DSL) and compiler framework called Spatial which implements these abstractions to support higher level, performance-oriented hardware accelerator design. Figure 1 shows an example of a basic implementation of matrix multiplication in Spatial. As this figure shows, Spatial code is like existing HLS languages in that programs are untimed and the language encourages accelerator designs to be expressed in terms of nested loops. However, unlike existing HLS tools, Spatial gives users more explicit control over the memory hierarchy through a library of on-chip and off-chip memory templates (e.g. the `DRAM` and `SRAM` in Figure 1). Spatial automatically

pipelines arbitrarily nested loops, and banks, buffers, and duplicates memories for the user based on parallel access patterns by default. This is in contrast to modern HLS tools, which largely rely on the user to add explicit pragmas to their code in order make these optimizations. Spatial also supports tuning of parameterized designs via automated design space exploration (DSE). Unlike prior approaches [22] which use variance-prone heuristic random search, Spatial employs an active machine learning framework called HyperMapper [11] to drive exploration. This tuning allows a single accelerator design to be quickly ported across target architectures and vendors with ease.

When targeting FPGAs, Spatial generates optimized, synthesizable Chisel code along with C++ code which can be used on a host CPU to administrate initialization and execution of the accelerator on the target FPGA. Spatial currently supports Xilinx Ultrascale+ VU9P FPGAs on Amazon's EC2 F1 Instances, Xilinx Zynq-7000 and Ultrascale+ ZCU102 SoCs, and Altera DE1 and Arria 10 SoCs. The constructs in Spatial are general across reconfigurable architectures, meaning Spatial programs can also be used to target CGRAs. In this paper, we demonstrate this by targeting our recently proposed Plasticine CGRA [32].

The contributions of this paper are as follows:

- We discuss the abstractions required to describe target-agnostic accelerator designs for reconfigurable architectures (Section 2). We then describe Spatial's implementation of these constructs (Section 3) and the optimizations that these abstraction enables in the Spatial compiler (Section 4).
- We describe an improved method of fast, automated design parameter space exploration using HyperMapper (Section 4). This approach is evaluated in Section 5.
- We evaluate Spatial's ability to efficiently express a wide variety of applications and target multiple architectures from the same source code. We demonstrate Spatial targeting two FPGAs and the Plasticine CGRA. We quantitatively compare Spatial to SDAccel on the VU9P FPGA on a diverse set of benchmarks (Section 5), showing a geometric mean speedup of 2.9× with 42% less code. We provide a qualitative comparison of Spatial to other related work in Section 6.

2 Language Criteria

It is critical for a language with the purpose of abstracting hardware design to strike the right balance between high-level constructs for improving programmer productivity and low-level syntax for tuning performance. Here, we motivate our discussion of Spatial by outlining requirements for achieving a good balance between productivity and achievable performance.

Control For most applications, control flow can be expressed in abstract terms. Data-dependent branching (e.g. if-statements) and nested loops are found in almost all applications, and in the common case these loops have a statically calculable initiation interval. These loops correspond to hierarchical pipelines which can be automatically optimized by the compiler in the majority of cases. The burden for specifying these control structures should therefore fall on the compiler, with the user intervening only when the compiler lacks information to automatically optimize the loop schedule.

Memory Hierarchy On most reconfigurable architectures, there are at least three levels of memory hierarchy: off-chip (DRAM), on-chip scratchpad (e.g. "block RAM" on FPGAs), and registers. Unlike CPUs, which present their memory as a uniformly accessible address space, reconfigurable architectures require programmers to explicitly manage the memory hierarchy. Previous languages like Sequoia [16] have demonstrated the benefits of explicit notions of memory hierarchy to programming language design. Moreover, loop unrolling and pipelining are essential for performance and area utilization, but these optimizations require on-chip memories to be partitioned, banked, and buffered to supply the bandwidth necessary for concurrent accesses. These decisions are made by statically analyzing memory access patterns with respect to loop iterators. The accelerator design language should therefore give the user a view of the target memory hierarchy and should include notions of loop iterators to enable automatic memory partitioning, banking, and buffering decisions.

In addition to on-chip memory management, accelerator designs must also explicitly administer transfers between off-chip and on-chip memories. This entails creating a soft memory controller which manages the off-chip memory. These memory controller implementations vary widely across different target architectures and vendors. However, common across these architectures is the need to optimize the memory controller based on access pattern. Unpredictable, data-dependent requests require more specialized memory controller logic than predictable, linear accesses. Instead of focusing on target-specific details, the language should allow users to focus on optimizing each transfer based on its access pattern. The accelerator language should therefore abstract these transfers as much as possible, while also giving constructs which specialize based on access patterns.

Host Interfaces Spatial architectures are commonly used as offload application accelerators. In this execution

model, the host generally allocates memory, prepares data structures, and interfaces with larger heterogeneous networks to receive and send data. Once data is prepared, the host invokes the accelerator and either waits for completion ("blocking" execution) or interacts with the perpetually running accelerator in a polling or interrupt manner ("non-blocking" execution). While management of communication and accelerator execution are commonly supported, the associated libraries and function calls vary widely across platforms and vendors, making code difficult to port or compare. For communication with the CPU host, a higher level language for accelerator design should provide constructs which abstract away the target architecture as much as possible.

Design Space Exploration As with any hardware design, accelerator design spaces can be extremely large and cumbersome to explore. While making optimizations like loop pipelining and memory banking automatic help to improve productivity, these transformations leave the compiler with numerous choices about how to allocate resources. These decisions can accumulate large performance/area tradeoff spaces which combine exponentially with application complexity. In a fixed implementation of general matrix multiplication, there is a large design space that includes the dimensions of on-chip tiles that hold portions of the full matrices and decisions about the parallelizations of loops that iterate over tiles as well as loops that iterate within these tiles. The parameters shown in lines 17 – 21 of Figure 1 expose just a few of these many design space parameters. Previous work [22] has shown how making the compiler aware of design parameters like pipelining, unrolling factors, and tile sizes can be used to speed up and automate parameter space exploration. Abstract hardware languages should therefore include both language and compiler support for design space parameters.

3 The Spatial Language

Spatial is a domain specific language for the design of accelerators implemented on reconfigurable spatial architectures, including FPGAs and CGRAs. The aim of the language is to simplify the accelerator design process, allowing domain experts to quickly develop, test, optimize, and deploy hardware accelerators, either by directly implementing high-level hardware designs or by targeting Spatial from another, higher level language.

In this section, we describe the abstractions Spatial includes to achieve a balance between productivity and performance-oriented detail. While space does not permit a full specification of the language, Table 1 provides an overview of the core subset of Spatial's syntax.

3.1 Control Structures

Spatial provides a mix of control structures which help users to more succinctly express their programs while also allowing the compiler to identify parallelization opportunities. These structures can be arbitrarily nested without restriction, allowing users to easily define hierarchical pipelines and nested parallelism. Table 1a provides a list of some of the control structures in the language. In addition to `Foreach` loops and state machines, Spatial also borrows ideas from parallel patterns [35, 39] to provide succinct functional syntax for reductions. While it is possible to express reductions in a purely imperative way, `Reduce` informs the compiler that the reduction function can be considered associative. Similarly, reduction across a series of memories using `MemReduce` exposes more levels of parallelism than an imperative implementation. For example, in Figure 1, the `MemReduce` on line 45 allows the compiler to parallelize over parameter `PAR_K`. This will result in multiple `tileC` tiles being populated in parallel, followed by a reduction tree to combine them into the accumulator `accum`.

`Foreach`, `Reduce`, and `MemReduce` can be parallelized by setting parallelization factors on their respective counters. When loop parallelization is requested, the compiler analyzes whether loop parallelization guarantees equivalent behavior to sequential execution. If this check fails, the compiler will issue an error. Spatial guarantees that a parallelized body will complete in its entirety before the next parallelized iteration is started, but makes no guarantees about the relative timing of operations across a single batch of unrolled iterations.

The bodies of Spatial control structures are untimed. The compiler automatically schedules operations, with the guarantee that functional behavior will not be changed. The schedule selected by the compiler can be pipelined, sequential, or streaming execution. In pipelined execution, the execution of loop iterations are overlapped. In innermost loops, the degree of overlap is based on the controller's average initiation interval. In outer loops, the amount of overlap is determined by the controller's "depth". Depth is defined as the maximum number of outer loop iterations a stage is allowed to execute before its consumer stages begin execution.

In sequential execution, a single iteration of a loop body is executed in its entirety before the next iteration begins. Sequential scheduling is equivalent to pipelining with the initiation interval equal to the loop body's latency, or, for outer controllers, a depth of 1. Streaming execution overlaps stages further by allowing each inner controllers to run asynchronously when inputs are available. Streaming is only a well-defined control scheme when communication between controllers is done through either streaming interfaces or queues.

Table 1. A subset of Spatial's syntax. Square brackets (e.g. `[T]`) represent a template's type parameter. Parameters followed by a '+' denote arguments which can be given one or more times, while a '⋆' denotes that an argument is optional. `DRAMs`, `Foreach`, `Reduce`, and `MemReduce` can all have arbitrary dimensions.

(a) Control Structures

`min⋆ until max by stride⋆ par factor⋆`
A counter over [**min**,**max**) ([0,**max**) if **min** is unspecified).
stride: optional counter stride, default is 1
factor: optional counter parallelization, default is 1

FSM`(init){`**continue**`}{`**action**`}{`**next**`}`
An arbitrary finite state machine, similar to a *while* loop.
init: the FSM's initial state
continue: the "while" condition for the FSM
action: arbitrary expression, executed each iteration
next: function calculating the next state

Foreach`(counter+){`**body**`}`
A parallelizable *for* loop.
counter: counter(s) defining the loop's iteration domain
body: arbitrary expression, executed each loop iteration

Reduce`(accum)(counter+){`**func**`}{`**reduce**`}`
A scalar reduction loop, parallelized as a tree.
accum: the reduction's accumulator register
counter: counter(s) defining the loop's iteration domain
func: arbitrary expression which produces a scalar value
reduce: associative reduction between two scalar values

MemReduce`(accum)(counter+){`**func**`}{`**reduce**`}`
Reduction over addressable memories.
accum: an addressable, on-chip memory for accumulation
counter: counter(s) defining the loop's iteration domain
func: arbitrary expression returning an on-chip memory
reduce: associative reduction between two scalar values

Stream`(⋆){`**body**`}`
A streaming loop which never terminates.
body: arbitrary expression, executed each loop iteration

Parallel`{`**body**`}`
Overrides normal compiler scheduling. All statements in the body are instead scheduled in a *fork-join* fashion.
body: arbitrary sequence of controllers

DummyPipe`{`**body**`}`
A "loop" with exactly one iteration.
Inserted by the compiler, generally not written explicitly.
body: arbitrary expression

(b) Optional Scheduling Directives

Sequential`.(`**Foreach**`|`**Reduce**`|`**MemReduce**`)`
Sets loop to run sequentially.

Pipe`(ii⋆).(`**Foreach**`|`**Reduce**`|`**MemReduce**`)`
Sets loop to be pipelined.
ii: optional overriding initiation interval

Stream`.(`**Foreach**`|`**Reduce**`|`**MemReduce**`)`
Sets loop to be streaming.

(c) Shared Host/Accelerator Memories

ArgIn`[T]`
Accelerator register initialized by the host

ArgOut`[T]`
Accelerator register visible to host after accelerator execution

HostIO`[T]`
Accelerator register the host may read and write at any time.

DRAM`[T](dims+)`
Burst-addressable, host-allocated off-chip memory.

(d) External Interfaces

StreamIn`[T](bus)`
Streaming input from a **bus** of external pins.

StreamOut`[T](bus)`
Streaming output to a **bus** of external pins.

(e) Host Interfaces

Accel`{`**body**`}`
A blocking accelerator design.

Accel`(⋆){`**body**`}`
A non-blocking accelerator design.

(f) Design Space Parameters

`default (min,max)`
`default (min,stride,max)`
A compiler-aware design parameter with given **default** value.
DSE explores the range [**min**, **max**] with optional **stride**.

3.2 Memories

Spatial offers a variety of memory templates that enable the user to abstractly but explicitly control allocation of data across an accelerator's heterogeneous memory. The Spatial compiler is aware of all of these memory types and is able to automatically optimize each of them.

Spatial's "on-chip" memories represent the creation of statically sized, logical memory spaces. Supported memory types include read-only lookup-tables (`LUTs`), scratchpads (`SRAM`), line buffers (`LineBuffer`), fixed size queues and stacks (`FIFO` and `LIFO`), registers (`Reg`), and register files (`RegFile`). These memories are always allocated using resources on the accelerator, and by default are not accessible by the host. While each memory is guaranteed to appear coherent to the programmer, the number and type of resources used to implement each memory is not restricted. With the exception of `LUTs` and `Regs` with explicit initial values, the contents of a memory is undefined upon allocation. These rules give the Spatial compiler maximum freedom to optimize memory access latency and resource utilization in the context of the entire application. Depending upon access patterns, the compiler may automatically duplicate, bank, or buffer the memory, provided the behavior of the final memory is unchanged.

"Shared" memories are allocated by the host CPU and accessible by both the host and the accelerator. These memories are typically used in the offload model to transfer data between the host and the accelerator. `DRAM` templates represent the slowest, largest level of the hierarchy. To help users optimize memory controllers, `DRAM` is read and written using explicit transfers to and

from on-chip memories. These transfers are specialized for predictable (`load` and `store`) and data-dependent (`scatter` and `gather`) access patterns.

3.3 Interfaces

Spatial offers several specialized interfaces for communication with the host and other external devices connected to the accelerator. Like memory templates, Spatial is capable of optimizing operations on these interfaces.

`ArgIn`, `ArgOut`, and `HostIO` are specialized registers with memory mappings on the CPU host. `ArgIns` may only be written by the host during device initialization, while `ArgOuts` can only be read, not written, by the host. `HostIO` can be read or written by the host at any time during accelerator execution. Additionally, scalars, including `DRAM` sizes, implicitly create `ArgIn` instances when used within an `Accel` scope. For instance, in Figure 1, the dimensions of matrices `A`, `B`, and `C` are passed to the accelerator via implicit `ArgIn`s since they are used to generate loop bounds (e.g. `A.rows`, `B.cols`).

`StreamIn` and `StreamOut` in Spatial are used to create connections to external interfaces. Streams are created by specifying a bus of input/output pins on the target device. Connection to external peripherals is done in an object-oriented manner. Every available Spatial target defines a set of commonly used external buses which can be used to allocate a `StreamIn` or `StreamOut`.

Spatial allows users to write host and accelerator code in the same program to facilitate communication between the two devices. The language's data structures and operations are classified as either "acceleratable" or "host"; only acceleratable operations have a defined mapping onto spatial architectures. Spatial makes this distinction in order to give users structure their algorithm in a way that is best for a reconfigurable architecture. Programs which heavily rely on dynamic memory allocation, for example, generally do not perform well on reconfigurable architectures, but can often be transformed at the algorithm level to achieve better performance.

Spatial programs explicitly partition work between the host and the accelerator using the `Accel` scope. As shown in Table 1e, these calls are specified as either blocking or non-blocking. Figure 1 shows an example of a blocking call, in which the product of two matrices is computed in the accelerator and then passed to the host only after it is completed. All operations called within this scope will be allocated to the targeted hardware accelerator, while all outside will be allocated to the host. Because of this, all operations within the `Accel` scope must be acceleratable.

Operations on the host include allocation of memory shared between the host and accelerator, transferring data to and from the accelerator, and accessing the host's file system. Arrays are copied to and from shared memory through `DRAM` using operations like `sendMatrix` and `getMatrix` shown in Figure 1. Scalars are transferred via `ArgIn` and `ArgOut` using `setArg` and `getArg`.

```
1  def FIR_Filter(args: Array[String]) {
2    val input   = StreamIn[Int](target.In)
3    val output  = StreamOut[Int](target.Out)
4    val weights = DRAM[Int](32)
5    val width   = ArgIn[Int]
6    val P = 16 (1,1,32)
7    // Initialize width with the first console argument
8    setArg(width, min(32, args(0).to[Int]) )
9    // Transfer weights from the host to accelerator
10   sendArray(weights, loadData[Int]("weights.csv"))
11
12   Accel {
13     val wts = RegFile[Int](32)
14     val ins = RegFile[Int](32)
15     val sum = Reg[Int]
16     // Load weights from DRAM into local registers
17     wts load weights(0::width)
18
19     Stream(*) {  // Stream continuously
20       // Shift in the most recent input
21       ins <<= input
22
23       // Create a reduce-accumulate tree with P inputs
24       Reduce(sum)(0 until width par P){i =>
25         wts(i) * ins(i)
26       }{(a,b) => a + b }
27
28       // Stream out the computed average
29       output := sum / width
30     }
31   }
32 }
```

Figure 2. A finite impulse response (FIR) filter.

After Spatial compilation, host operations are code generated to C++. From the host's perspective, the `Accel` scope doubles as a black box for generating target-specific library calls to run the accelerator. This syntax serves to completely abstract the tedious, target-specific details of initializing and running the accelerator.

Spatial currently assumes that the system has one target reconfigurable architecture. If the program defines multiple `Accel` scopes, these are loaded and run sequentially in declaration order. However, this constraint can easily be relaxed in future work.

3.4 Parameters

Parameters in Spatial are created using the syntax shown in Table 1f. Since each parameter must have a fixed value by the time the compiler generates code, the supplied range must be statically computable. Parameters can be used to specify the dimensions of addressable on-chip memories and DRAMs. They can also be used when creating counters to specify a parameterized step size or parallelization factor, or when specifying the pipelining depth of outer controllers. An application's implicit and explicit application parameters together define a design space which the compiler can later automatically explore.

3.5 Examples

We conclude discussion of the Spatial language with two examples. Figure 2 shows a streaming implementation of a finite impulse response (FIR) filter. This example

```
1  def Merge_Sort(offchip: DRAM[Int], offset: Int) {
2    val N = 1024  // Static size of chunk to sort
3    Accel {
4      val data  = SRAM[Int](N)
5      data load offchip(offset::N+offset)
6
7      FSM(1){m => m < N}{ m =>
8        Foreach(0 until N by 2*m){ i =>
9          val lower = FIFO[Int](N/2).reset()
10         val upper = FIFO[Int](N/2).reset()
11         val from  = i
12         val end   = min(i + 2*m - 1, N) + 1
13
14         // Split data into lower and upper FIFOs
15         Foreach(from until i + m){ x =>
16           lower.enq(data(x))
17         }
18         Foreach(i + m until end){ y =>
19           upper.enq(data(y))
20         }
21
22         // Merge of the two FIFOs back into data
23         Foreach(from until end){ k =>
24           val low  = lower.peek() // Garbage if empty
25           val high = upper.peek() // Garbage if empty
26           data(k) = {
27             if      (lower.empty) { upper.deq() }
28             else if (upper.empty) { lower.deq() }
29             else if (low < high)  { lower.deq() }
30             else                  { upper.deq() }
31           }
32         }
33       }
34     }{ m => 2*m /* Next state logic */ }
35
36     offchip(offset::offset+N) store data
37   }
38 }
```

Figure 3. Part of a design for in-place merge sort.

demonstrates how, when using `Stream(*)`, Spatial's semantics are similar to other dataflow-oriented streaming languages. The body of the loop on line 24 is run each time a valid element appears at the `StreamIn` input. Spatial pipelines this body to maximize its throughput.

While basic FIR filters are simple to write and tune even in HDLs, Spatial makes expanding upon simple designs easier. The number of weights and taps in this example can be set at device initialization, without having to resynthesize the design. Additionally, the number of elements combined in parallel in the filter is defined as a parameter. Design space exploration can automatically tune the design for the smallest area or lowest latency.

Figure 3 shows a simple implementation of a fixed size merge sort in Spatial. Here, data is loaded into on-chip scratchpad, sorted, and then stored back into main memory. The language's distinction between on-chip and off-chip memory types makes writing and reasoning about tiled designs like this one much more natural. This implementation uses a statically sized `SRAM` and two `FIFOs` to split and order progressively larger size chunks of the local data. The chunk size is determined by the outermost loop on line 8, and increments in powers of two, which is best expressed in Spatial as an FSM.

4 The Spatial Compiler

The Spatial compiler provides source-to-source translations from applications in the Spatial language to synthesizable hardware descriptions in Chisel RTL [9]. In this section, we describe the compiler's intermediate representation and its key passes, as summarized in Figure 4. Apart from chisel generation, these passes are common to targeting both FPGAs and the Plasticine CGRA. Details of targeting Plasticine are discussed in prior work [32].

Intermediate Representation Spatial programs are internally represented in the compiler as a hierarchical dataflow graph (DFG). Nodes in this graph represent control structures, data operations, and memory allocations, while edges represent data and effect dependencies. Nesting of controllers directly translates to the hierarchy in the intermediate representation. Design parameters are kept as graph metadata, such that they can be independently updated without changing the graph itself.

When discussing DFG transformations and optimizations, it is often useful to think about the graph as a controller/access tree. Figure 5 shows an example of one such controller tree for the memory `tileB` in the Spatial code example in Figure 1. Note that transfers between on-chip and off-chip memory expand to a control node which linearly accesses the on-chip memory, in this case by iterators `e` and `f`. This tree abstracts away most primitive operations, leaving only relevant controller hierarchy and the memory accesses for a specific memory.

Within the acceleratable subset of Spatial, nodes are formally separated into three categories: control nodes, memory allocation nodes, and primitive nodes. Control nodes represent state machine structures like `Foreach` and `Reduce` described in Section 3.1. Primitive nodes are operations which may consume, but never produce, control signals, including on-chip memory accesses. Primitive nodes are further broken down into "physical" operations requiring resources and "ephemeral" operations which are only used for bookkeeping purposes in the compiler. For example, bit selects and grouping of words into structs require no hardware resources but are used to track necessary wires in the generated code.

Control Insertion To simplify reasoning about control signals, Spatial requires that control nodes do not contain both physical primitive nodes and other control nodes. The exception to this rule is conditional `if` statements, which can be used in the same scope as primitives as long as they contain no control nodes but conditionals themselves. This requirement is satisfied by a DFG transformation which inserts `DummyPipe` control nodes around primitive logic in control bodies which also contain control nodes. The `DummyPipe` node is a bookkeeping control structure which is logically equivalent to a loop with exactly one iteration. Thereafter, control nodes with primitive nodes are called "inner" control nodes, while controllers which contain other nested controllers are called "outer" nodes.

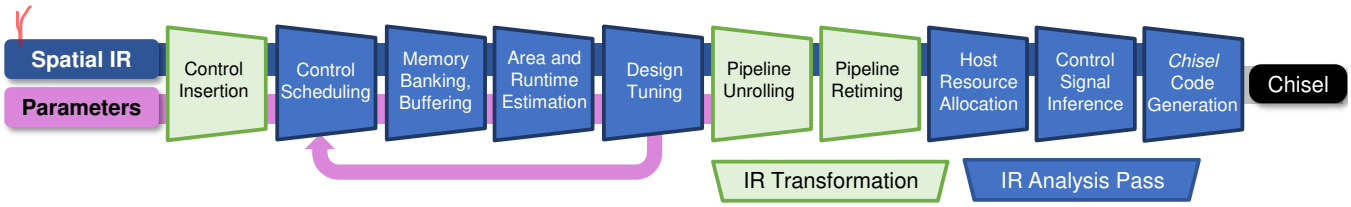


Figure 4. A summary of the passes in the Spatial compiler for targeting FPGAs.

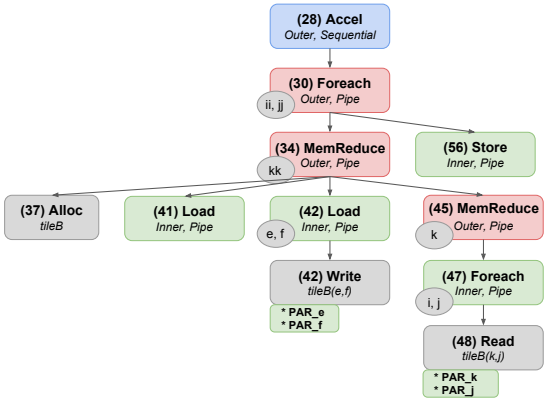


Figure 5. The control/access tree for the `SRAM tileB` in the matrix multiply example in Figure 1.

Controller Scheduling After controller insertion, the compiler will then schedule the operations within each controller. By default, the compiler will always attempt to pipeline loops regardless of nesting level. The behavior of the compiler's scheduler can be overridden by the user using the directives listed in Table 1b.

Inner pipeline schedules are based on their initiation interval. The compiler first collects resource initiation intervals for each primitive node in the given controller based on an internal, target-dependent lookup table. Most primitive operations are pipelined for a resource initiation interval of 1. The compiler then calculates all loop carried dependencies within the pipeline based on the dataflow graph. For non-addressable memories, the total initiation interval is the maximum of path lengths between all dependent reads and the writes. For addressable memories, the path length of loop carried dependencies is also multiplied by the difference in write and read addresses. If the addresses are loop-independent, the initiation interval is the path length if they may be equal, and 1 if they are provably not equal. If the distance between the addresses cannot be determined statically, the initiation interval is infinite, meaning the loop must be run sequentially. The final initiation interval of the controller is defined as the maximum of the initiation intervals of all loop carried dependencies and all resource initiation intervals.

The compiler also attempts to pipeline the bodies of outer control nodes in a similar manner, but computes dataflow scheduling in terms of inner control nodes and number of stages rather than primitive nodes and cycles. For example, the outer `MemReduce` in line 34 of Figure 1 contains 4 sub-controllers: the load into `tileA` (line 41), the load into `tileB` (42), the inner `MemReduce` (45), and an reduction stage combining intermediate tiles (53). Based on data dependencies, the compiler infers that the two loads can be run in parallel, followed by the inner `MemReduce` and the tile reduction. It will also determine that multiple iterations of this outer loop can also be pipelined through these stages.

Memory Analysis Loop parallelization only serves to improve performance if there is sufficient on-chip bandwidth to feed the duplicated computation. Spatial's memory analysis banks and buffers on-chip memories to maximize this available on-chip read and write bandwidth. Memory banking, also called data partitioning, is the process of dividing a memory's address space across multiple physical instances in order to create additional ports for concurrent accesses within the same controller. Partitioning is possible when the access patterns are statically predictable and guaranteed to never conflict access the same port/bank. While a single port can be time multiplexed, this entirely negates the benefits of parallelization by increasing the whole pipeline's required initiation interval. Note that while banking can trivially be achieved by memory duplication, Spatial aims to also minimize the total amount of memory resources.

Spatial leverages the memory partitioning strategy based on conflict polytope emptiness testing described by Wang et. al. [41]. We extend this strategy by accounting for random access patterns and memory accesses across nested loops. Random accesses are modeled as additional dimensions in the conflict polytope as if they were additional loop iterators. Spatial minimizes the number of random access symbols used in this way by identifying affine combinations of random values. For example, an access to a memory at address x and $x+1$ only requires one random variable, x, as the second is a predictable, affine function of the first. Spatial also supports banking per dimension to account for cases where only some dimensions are accessed predictably.

Non-addressed memories like `FIFOs` and `FILOs` are modeled as addressed memories. Each access to these

memory types is represented as a linear access of all loop iterators around the memory access relative to the memory's definition. Spatial forbids parallelization of outer loops around non-addressed accesses, as this violates the guarantee of equivalent behavior to sequential execution.

To handle multiple pipelined accesses across stages within an outer loop, Spatial also automatically buffers on-chip memories. Buffering creates multiple copies of the same memory for maintaining versions of the data across overlapped loop iterations. Without this optimization, pipeline parallel accesses to the same memory across different stages of a coarse-grain pipeline would not be able to run concurrently. See Appendix A.1 for details on how both banking and buffering are computed.

For example, as shown in Figure 5, `tileB` has two parallelized accesses, the load on line 42 and the read on line 48. If all (implicit and explicit) parallelization factors are set to 2, this corresponds to 4 accesses per loop. Spatial then builds the access polytope corresponding to all accesses in each loop, and determines the banking strategy that works for both loops. In this example, this means the SRAM will be banked such that each element within a 2x2 square will reside in a different physical bank to allow fully parallel access. If the `MemReduce` on line 34 is pipelined, `tileB` will be double buffered to protect the reads (line 48) in one iteration of the outer loop from the writes (line 42) in the next iteration.

Area and Runtime Estimation Spatial evaluates a given set of parameters by running a pair of estimation passes to approximate the area and runtime of the application. These passes are driven by analytical resource and runtime models similar to those used in our prior work on the Delite Hardware Definition Language (DHDL) [22], but Spatial expands this approach to account for streaming throughput, arbitrary control flow, and finite state machines. Both runtime and area utilization models are built from a set of about 2000 one-time characterization runs on each target platform.

Design Space Exploration The scheduling and memory banking options identified by the compiler, together with loop parallelization and tile size parameters, forms a design space for the application. The design tuning pass is an optional compiler pass which allows for fast exploration of this design space in order to make area/runtime design tradeoffs. When design tuning is enabled, it repeatedly picks design points and evaluates them by rerunning the control scheduling, memory analysis, and estimation analysis passes. The output from this search is a single set of parameters from the Pareto frontier.

Unfortunately, application design spaces tend to be extremely large, and exhaustive search on an entire space is often infeasible. Of the benchmarks discussed in Section 5, only BlackScholes has a relatively small space of about $80,000$ points. While this space can be explored exhaustively by Spatial in a few minutes, other spaces are much larger, spanning 10^6 to 10^{10} points and taking hours or days to exhaustively search. For example, even with the few explicit design parameters exposed in the code in Figure 1, when combined with implicit pipelining and parallelization parameters, this code already has about 2.6×10^8 potential designs. DHDL [22] employed random search after heuristic pruning, reducing the total space by two to three orders of magnitude. However, this approach has high variance on larger design spaces and may inadvertently prune desirable points.

To reduce the variance on larger design spaces, Spatial's design space exploration flow integrates an active learning-based autotuner called HyperMapper [11, 27, 36]. HyperMapper is a multi-objective derivative-free optimizer (DFO), and has already been demonstrated on the SLAMBench benchmarking framework [28]. HyperMapper creates a surrogate model using a Random Forest regressor, and predicts the performance over the parameter space. This regressor is initially built using only few hundred random design point samples and is iteratively refined in subsequent active learning steps.

Unrolling Following selection of values for design parameters, Spatial finalizes these parameters in a single graph transformation which unrolls loops and duplicates memories as determined by prior analysis passes. `Reduce` and `MemReduce` patterns are also lowered into their imperative implementations, with hardware reduction trees instantiated from the given reduction function. The two `MemReduce` loops in Figure 1, for example, will each be lowered into unrolled `Foreach` loops with explicitly banked memory accesses and explicitly duplicated multiply operations. The corresponding reduction across tiles (lines 52 – 53) are lowered into a second stage of the `Foreach` with explicit reduction trees matching the loop parallelization.

Retiming After unrolling, the compiler retimes each inner pipeline to make sure data and control signals properly line up and that the target clock frequency can be met. To do this, the compiler orders primitive operations within each pipeline based on effect and dataflow order. Based on this ordering, the compiler then inserts pipeline and delay line registers based on lookup tables which map each primitive node to an associated latency. Dependent nodes which have less than a full cycle of delay are kept as combinational operations, with a register only being inserted after the last operation. This maximizes the achievable clock frequency for this controller while also minimizing the required initiation interval.

Code Generation Prior to code generation, the compiler first allocates register names for every `ArgIn`, `ArgOut`,

and `HostIO`. In the final pass over the IR, the code generator then instantiates hardware modules from a library of custom, parameterized RTL templates written in Chisel and infers and generates the logic required to stitch them together. These templates include state machines that manage communication between the various control structures and primitives in the application, as well as the banked and buffered memory structures and efficient arithmetic operations. Finally, all generated hardware is wrapped in a target-specific, parameterized Chisel module that arbitrates off-chip accesses from the accelerator with the peripheral devices on the target FPGA.

5 Evaluation

In this section, we evaluate Spatial by comparing programmer productivity and the performance of generated designs to Xilinx's commercial HLS tool, SDAccel. We then evaluate the HyperMapper design tuning approach and demonstrate Spatial's advantages for portability across FPGAs and the Plasticine CGRA.

5.1 FPGA Performance and Productivity

We first evaluate the FPGA performance and productivity benefits of Spatial against SDAccel, a commercial C-based programming tool from Xilinx for creating high-performance accelerator designs. We use SDAccel in our study as it has similar performance and productivity goals as Spatial, supports the popular OpenCL programming model, and performs several optimizations related to loop pipelining, unrolling, and memory partitioning [42]. Baseline implementations of the benchmarks in Table 2 have been either obtained from a public SDAccel benchmark suite from Xilinx [45], or written by hand. Each baseline has then been manually tuned by using appropriate HLS pragmas [43] to pick loop pipelining, unrolling, and array banking factors, and to enable dataflow optimizations. Design points for Spatial are chosen using the DSE flow described in Section 4.

We measure productivity by comparing number of lines of source code used to describe the FPGA kernel, excluding host code. We measure performance by comparing runtimes and FPGA resources utilized for each benchmark on a Xilinx Ultrascale+ VU9P board with a fabric clock of 125 MHz, hosted on an Amazon EC2 F1 instance. We generate FPGA bitstreams targeting the VU9P architecture for each benchmark using both Spatial and SDAccel, and obtain resource utilization data from the post place-and-route reports. We then run and verify both designs on the FPGA and measure the execution times on the board. CPU setup code and data transfer time between CPU and FPGA is excluded from runtime measurements for both tools.

Table 2 shows the input dataset sizes and the full comparison between lines of source code, resource utilization, and runtime of the benchmarks implemented in SDAccel and Spatial. In terms of productivity, language constructs in Spatial like `load` and `store` for transferring dense sparse data from DRAM reduces code bloat and increases readability. Furthermore, by implicitly inferring parameters such as parallelization factors, Spatial code is largely free of annotations and pragmas.

Spatial achieves moderate speedups over SDAccel of 1.63× and 1.33× respectively on *BlackScholes* and *TPC-H Q6*. Both benchmarks stream data from DRAM through a deeply pipelined datapath which is amenable to FPGA acceleration. Dataflow support in SDAccel using the DATAFLOW pragma [44] and streaming support in Spatial allows both tools to efficiently accelerate such workloads. In *K-Means*, coarse-grained pipelining support allows Spatial to achieve roughly the same performance as SDAccel using 1.5× fewer BRAMs. Specialized DRAM scatter/gather support enables Spatial to achieve a 3.48× speedup on *PageRank*.

We see speedups of 8.48×, 1.37×, and 14.15× for compute-heavy workloads *GDA*, *GEMM*, and *SW*, respectively. The baseline for *SW* is implemented by Xilinx as a systolic array, while the Spatial implementation uses nested controllers. *GEMM* and *GDA* contain opportunities for coarse-grained pipelining that are exploited within Spatial. GDA, for example, contains an outer product operation, during which the data in the same buffer is repeatedly accessed and reused. While this operation can be pipelined with a preceding loop producing the array, SDAccel's DATAFLOW pragma does not support such access patterns that involve reuse. As a result, SDAccel requires larger array partitioning and loop unrolling factors to offset the performance impact, at the expense of consuming more FPGA BRAM. In addition, nested controllers in *GEMM* can be parallelized and pipelined independently in Spatial, while SDAccel automatically unrolls all inner loops if an outer loop is parallelized. Spatial can therefore explore design points that cannot be easily expressed in SDAccel. Finally, as the Spatial compiler performs analyses on a parameterized IR, the compiler can reason about larger parallelization factors without expanding the IR graph. SDAccel unrolls the graph as a preprocessing step, hence creating larger graphs when unrolling and array partitioning factors are large. This has a significant impact on the compiler's memory footprint and compilation times, making better designs difficult or impossible to find.

Spatial provides a productive platform to program FPGAs, with a 42% reduction in lines of code compared to SDAccel averaged across all benchmarks. On the studied benchmarks, Spatial achieves a geometric mean speedup of 2.9× compared to an industrial HLS tool.

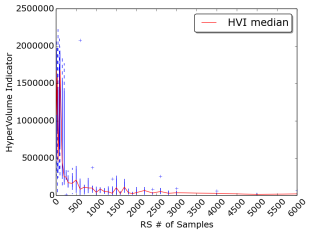
Table 2. Number of lines of code (LOC), area utilization, and runtime comparisons between SDAccel and Spatial on a single VU9P FPGA. For reference, the first row lists the total number of FPGA resources available. LOC improvements are percent improvements from SDAccel. The remaining improvement factors are calculated as (*SDAccel*/*Spatial*).

				LOC	LUTs	BRAM	DSPs	Time (ms)
Benchmark	**Data Sizes**	**DSE size**	***Capacity***	—	914400	1680	5640	—
BS			SDAccel	175	363368	550	290	6.18
Black-Scholes	960,000 options	7.7×10^4	Spatial	93	698885	493	945	3.79
Option pricing			**Improvement**	**46.8%**	**0.52×**	**1.12×**	**0.31×**	**1.63×**
GDA			SDAccel	73	356857	594	108	10.75
Gaussian discriminant analysis	1024 rows 96 dimensions	3.0×10^{10}	Spatial	64	378130	858	216	1.27
			Improvement	**12.3%**	**0.94×**	**0.69×**	**0.50×**	**8.48×**
GEMM			SDAccel	110	341894	674	206	1207.26
Matrix multiply	A: 1024×1024 B: 1024×1024	2.6×10^8	Spatial	44	426295	500	798	878.45
			Improvement	**60.0%**	**0.80×**	**1.35×**	**0.26×**	**1.37×**
KMeans			SDAccel	146	356382	657	539	73.04
K-Means clustering	200 iterations 320×32-element points	2.1×10^6	Spatial	81	369348	453	105	53.25
			Improvement	**44.5%**	**0.96×**	**1.45×**	**5.13×**	**1.37×**
PageRank			SDAccel	112	337102	549	17	2041.62
Node ranking algorithm	DIMACS10 Chesapeake 10000 iterations	4.1×10^3	Spatial	77	418128	862	81	587.35
			Improvement	**31.2%**	**0.81×**	**0.64×**	**0.21×**	**3.48×**
SW			SDAccel	240	541617	547	12	8.67
Smith-Waterman	256 base pairs	2.1×10^6	Spatial	82	330063	470	9	0.61
DNA alignment			**Improvement**	**65.8%**	**1.64×**	**1.16×**	**1.33×**	**14.15×**
TQ6			SDAccel	74	356978	548	15	18.61
TPC-H Q6	6,400,000 records	3.5×10^9	Spatial	48	472868	574	393	13.97
Filter reduction			**Improvement**	**35.1%**	**0.75×**	**0.95×**	**0.04×**	**1.33×**
Average			**Improvement**	**42.3%**	**0.87×**	**1.01×**	**0.42×**	**2.90×**

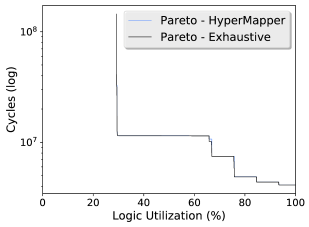
5.2 Design Space Exploration

We next perform a preliminary evaluation of HyperMapper for quickly approximating Pareto frontier over two design objectives: design runtime and FPGA logic utilization (LUTs). For this evaluation, we run HyperMapper with several seeds of initial random sample, with the number of samples R ranging from 1 to 6000 designs, and run 5 active learning iterations of at most 100 samples each. Heuristic search prunes using the heuristics defined in the DHDL work [22] and then randomly samples up to 100,000 points. For both approaches, design tuning takes up to 1 – 2 minutes, varying by benchmark complexity.

Figure 6a shows the hypervolume indicator (HVI) function for the *BlackScholes* benchmark as a function of the initial number of random samples. The HVI gives the area between the estimated Pareto frontier and the space's true Pareto curve, found from exhaustive search. By increasing the number of random samples to bootstrap the active learning phase, we see two orders of magnitude improvement in HVI. Furthermore, the overall variance goes down very quickly as the number of random samples increases. As a result, the autotuner is robust to randomness and only a handful of random samples are needed to bootstrap the active learning phase. As shown in Figure 6b, HyperMapper is able to reach a close approximation of the true Pareto frontier with less than 1500 design points.



(a) HyperMapper HVI versus initial random samples (R) five number summary.



(b) Exhaustive and HyperMapper (R=1000) generated Pareto curves.

Figure 6. Design space tuning on *BlackScholes*

On benchmarks like GDA with sparser design spaces, HyperMapper spends much of its time evaluating areas of the space with invalid designs which cannot fit on the FPGA. HyperMapper's accuracy for these benchmarks is consequently lower than the heuristic approach. Consequently, in future work, we plan to extend HyperMapper with a valid design prediction mechanism and evaluate this tuning approach on a wider class of benchmarks.

5.3 Spatial Portability

We next demonstrate the portability of Spatial code by targeting two different FPGA architectures; (1) the Zynq ZC706 SoC board, and (2) The Virtex Ultrascale+ VU9P on the Amazon EC2 F1. Designs on the VU9P use a single DRAM channel with a peak bandwidth of

Table 3. Runtimes (ms) of tuned designs on ZC706, followed by runtimes and speedup (×) of directly porting these designs to the VU9P, then runtimes and successive speedup over ported designs when tuned for the VU9P. The *Total* column shows the cumulative speedup.

FPGA	ZC706	VU9P				Total
Design	Tuned	Ported		Tuned		
	Time	Time	×	Time	×	×
BS	89.0	35.6	2.5	3.8	9.4	23.4
GDA	8.4	3.4	2.5	1.3	2.6	6.5
GEMM	2226.5	1832.6	1.2	878.5	2.1	2.5
KMeans	358.4	143.4	2.5	53.3	2.7	6.7
PageRank	1299.5	1003.3	1.3	587.4	1.7	2.2
SW†	1.3	0.5	2.5	0.5	1.0	2.5
TQ6	69.4	15.0	4.6	14.0	1.1	5.0

†SW with 160 base pairs, the largest to fit on the ZC706.

19.2 GB/s. The ZC706 is much smaller than the VU9P in terms of FPGA resource and has a smaller DRAM bandwidth of 4.26 GB/s. We target both the ZC706 and VU9P from the same Spatial code for all benchmarks listed in Table 2. Benchmarks are tuned for each target using target-specific models with automated DSE. Clock frequency is fixed at 125 MHz for both FPGAs.

Table 3 shows the speedups achieved on the VU9P over the ZC706. The results show that not only can the same Spatial source code be ported to architectures with different capabilities, the application can also be automatically tuned to better take advantage of resources in each target. Compute-bound benchmarks *BlackScholes*, *GDA*, *GEMM*, *K-Means* achieve speedups of up to 23× on the VU9P over the ZC706. Porting these designs to the VU9P alone has a 1.2× to 2.5× due to increased main memory bandwidth, but a majority of the benefit of the larger FPGA comes from tuning the parallelization factors to use more resources. While *SW* is also compute bound, the size of the dataset was limited by the smaller FPGA. In this case, the larger capacity of the VU9P does not improve runtime, but instead allows handling of larger datasets.

Memory-bound benchmark *TPC-H Q6* benefits from the higher DRAM bandwidth available on the VU9P. Porting this benchmark immediately gives a 4.6× runtime improvement from the larger main memory bandwidth, while further parallelizing controllers to create more parallel address streams to DRAM helps the application make better use of this bandwidth. *PageRank* is also bandwidth-bound, but the primary benefit on the VU9P comes from specializing the memory controller to maximize utilized bandwidth for sparse accesses.

Finally, we demonstrate the portability of Spatial beyond FPGA architectures by extending the compiler to map the Spatial IR to target our proposed Plasticine CGRA [32]. Plasticine is a two-dimensional array of compute (PCUs) and memory (PMUs) tiles with a statically configurable interconnect and address generators (AG) at the periphery to perform DRAM accesses. The Plasticine architecture is a significant departure from an FPGA, with more constraints on memory banking and computation, including fixed size, pipelined SIMD lanes.

Table 4. Plasticine DRAM bandwidth, resource utilization, runtime, and speedup (×) over VU9P FPGA.

App	Avg DRAM BW (%)		Resource Utilization (%)			Time (ms)	×
	Load	Store	PCU	PMU	AG		
BS	77.4	12.9	**73.4**	10.9	20.6	2.33	1.6
GDA	24.0	0.2	**95.3**	73.4	38.2	0.13	9.8
GEMM	20.5	2.1	**96.8**	64.1	11.7	15.98	55.0
KMeans	8.0	0.4	**89.1**	57.8	17.6	8.39	6.3
TQ6	**97.2**	0.0	29.7	37.5	**70.6**	8.60	1.6

We simulate Plasticine with a 16 × 8 array of 64 compute and 64 memory tiles, with a 1 GHz clock and a main memory with a DDR3-1600 channel with 12.8 GB/s peak bandwidth. Table 4 shows the DRAM bandwidth, resource utilization, runtime, and speedup of the Plasticine CGRA over the VU9P for a subset of benchmarks.

Streaming, bandwidth-bound applications like *TPC-H Q6* efficiently exploit about 97% of the available DRAM bandwidth. Compute-bound applications *GDA*, *GEMM*, and *K-Means* use around 90% of Plasticine's compute tiles. Plasticine's higher on-chip bandwidth also allows these applications to better utilize the compute resources, giving these applications speedups of 9.9×, 55.0×, and 6.3×. Similarly, the deep compute pipeline in *BlackScholes* occupies 73.4% of compute resources after being split across multiple tiles, giving a speedup of 1.6×.

6 Related Work

We conclude with a qualitative comparison of Spatial to related work, drawing from the criteria in Section 2.

HDLs Hardware description languages like Verilog and VHDL are designed for arbitrary circuit description. To achieve maximum generality, they require users to explicitly manage timing, control signals, and local memories. Loops are expressed by state machines in flattened RTL. One exception to this is Bluespec SystemVerilog [8], which supports state machine inference from nested while loops. Recent advancements in HDLs have largely been aimed at meta-programming improvements and increasing the size of hardware module libraries. Languages like Chisel [9], MyHDL [1] and VeriScala [23] make procedural generation of circuits simpler by embedding their HDL in a software language (e.g. Scala or Python). Similarly, Genesis2 [37] adds Perl scripting support to SystemVerilog to help drive procedural generation. While these improvements allow for more powerful meta-programming compared to Verilog `generate` statements, users still write programs at a timed circuit level.

Lime Lime is a Java-based programming model and runtime from IBM which aims to provide a single unified language to program heterogeneous architectures. Lime natively supports custom bit precisions and includes collection operations, with parallelism in such operations inferred by the compiler. Coarse-grained pipeline and data parallelism are expressed through "tasks". Coarse-grained streaming computation graphs can be constructed using built-in constructs like `connect`, `split`, and `join`. The Lime runtime system handles buffering, partitioning, and scheduling of stream graphs. However, coarse-grained pipelines which deviate from the streaming model are not supported, and the programmer has to use a low-level messaging API to handle coarse-grained graphs with feedback loops. Additionally, the compiler does not perform automatic design tuning. Finally, the compiler's ability to instantiate banked and buffered memories is unclear as details on banking multi-dimensional data structures for arbitrary access patterns are not specified.

HLS High-level synthesis tools such as LegUp [12], Vivado HLS [3], Intel's FPGA SDK for OpenCL [5], and SDAccel [42] allow users to write FPGA designs in C/C++ and OpenCL. Using these tools, applications can be expressed at a high level, in terms of arrays and untimed, nested loops. However, while inner loop pipelining, unrolling, and memory banking and buffering are done by the compiler, they generally require explicit user pragmas. While previous work has used polyhedral tools to automate banking decisions for affine accesses within a single loop nest [41], it does not address non-affine cases or cases where accesses to the same memory occur in multiple loop nests. While pragmas like Vivado HLS's *DATAFLOW* enable limited support for pipelining nested loops, pipelining at arbitrary loop nest levels is not yet supported [2]. Tools like Aladdin [38] have also been created to help automate the process of tuning the pragmas in HLS programs, but designs in HLS still require manual hardware optimization [26].

MaxJ MaxJ is a proprietary language created by Maxeler which allows users to express dataflow algorithms in Java libraries, emphasizing timing at the level of "ticks" of valid streaming elements rather than cycles. [24]. Users must fall back to flattened, HDL-like syntax for state machines when writing nested loops. Memories are inferred based on relative stream offsets, which, while convenient for stream processing, hides hardware implementation details from the user which could otherwise help drive optimization. Additionally, MaxJ currently can only be used to target supported Maxeler platforms.

DHDL The Delite Hardware Description Language (DHDL) [22] is a precursor to Spatial, in that it allows programmers to describe untimed, nested, parallelizable hardware pipelines and compile these to hardware. While DHDL supports compiler-aware design parameters and automatic design tuning, it has no support for data-dependent control flow, streaming, or memory controller specialization. DHDL also has no support for generalized memory banking or buffering and relies on its backend, MaxJ, for retiming and initiation interval calculation.

Image Processing DSLs Recently proposed image processing DSLs provide high-level specifications for targeting various accelerator platforms, including GPUs and FPGAs.The narrow domain allows these DSLs to offer more concise abstractions for specifying stencil operations. When targeting accelerators, these languages usually rely on source-to-source translation. $HIPA^{CC}$ [25], for example, uses a source-to-source compiler from a C-like front-end to generate CUDA, OpenCL, and Renderscript for targeting GPUs. Recent work on Halide [35] has demonstrated targeting heterogeneous systems, including the Xilinx Zynq's FPGA and ARM cores, by generating intermediate C++ and Vivado HLS [33]. Rigel [20] and Darkroom [19] generate Verilog, and PolyMage [14] generates OpenMP and C++ for high-level synthesis. Rigel and Darkroom support generation of specialized memory structures on FPGAs, such as line buffers, in order to capture reuse. $HIPA^{CC}$ can infer memory hierarchy on GPUs from a fix set of access patterns. These DSLs capture parallelism within a given stencil, typically across image channels and across the image processing pipeline.

Compared to image processing DSLs, Spatial is more general and provides a lower level of abstraction. Spatial can express pipelining and unrolling for arbitrary loop hierarchies and explicitly exposes the memory hierarchy while automatically banking, buffering, and duplicating structures for arbitrary access patterns. These features, along with Spatial's design tuning capabilities, make Spatial a natural fit as an optimizing backend target for image processing DSLs.

7 Conclusion

In this work, we presented Spatial, a new domain-specific DSL for the design of application accelerators on reconfigurable architectures. Spatial includes hardware-specific abstractions for control, memory, and design tuning which help to provide a balance between productive and performance-driven accelerator design. We have demonstrated that Spatial can target a range of architectures from a single source, and can achieve average speedups of 2.9× over SDAccel with 42% less code.

The Spatial language and compiler is an ongoing, open source project at Stanford. Related documentation and releases can be found at https://spatial.stanford.edu.

```
1  function GroupAccesses:
2    input: A → set of reads or writes to m
3    G = ∅ set of sets of compatible accesses
4    for all accesses a in A:
5      for all sets of accesses g in G:
6        if IComp(a, a′) for all a′ in g then
7          add a to g
8          break
9        else add {a} to G
10   return G
11 end function
12
13 function ConfigureMemory:
14   input: A_r → set of reads of m
15   input: A_w → set of writes to m
16   G_r = GroupAccesses(A_r)
17   G_w = GroupAccesses(A_w)
18   I = ∅ set of memory instances
19   for all read sets R in G_r:
20     I_r = {R}
21     I_w = ReachingWrites(G_w, I_r)
22     i = BankAndBuffer(I_r, I_w)
23     for each inst in I:
24       I′_r = ReadSets[inst] + R
25       I′_w = ReachingWrites(G_w, I′_r)
26       if OComp(A_1,A_2) ∀A_1 ≠ A_2 ∈ (G_w ∪ I′_r) then:
27         i′ = BankAndBuffer(I′_r, I′_w)
28         if Cost(i′) < Cost(i) + Cost(inst) then:
29           remove inst from I
30           add i′ to I
31           break
32     if i has not been merged then add i to I
33   return I
34 end function
```

Figure 7. Banking and buffering algorithm for calculating instances of on-chip memory m.

A Appendix

A.1 Memory Banking and Buffering

Figure 7 gives pseudocode for Spatial's algorithm to bank and buffer accesses to a given memory m across all loop nests. We group read and write accesses into "compatible" which occur and parallel but which can be banked together (lines 1 – 12). Two accesses a_1 and a_2 within iteration domains D_1 and D_2 are banking compatible ($IComp$) if

$$IComp(a_1, a_2) = \nexists\, \vec{i} \in (D_1 \cup D_2)\ s.t.\ a_1(\vec{i}) = r_2(\vec{i})$$

where an iteration domain is the space of values of all loop iterators and $a(i)$ is the multi-dimensional address corresponding to access a for iterator values i. This check can be implemented using a polytope emptiness test.

After grouping, each group could be directly mapped to a physical "instance", or copy, of m. However, to minimize required resources, we greedily merge groups together (lines 19 – 32). Merging is done when the cost of a merged instance is less than the cost of adding a separate, coherent instance for that group. Two sets of accesses A_1 and A_2 allow merging ($OComp$) if

$$OComp(A_1, A_2) = \nexists\, (a_1 \in A_1, a_2 \in A_2)\ s.t.$$
$$LCA(a_1, a_2) \in Parallel \cup (Pipe \cap Inner)$$

where *Parallel*, *Pipe*, and *Inner* are the set of Parallel, pipelined, and inner controllers in the program, respectively. If this condition holds, all accesses between the two instances either occur sequentially or occur as part of a coarse-grain pipeline. Sequential accesses can be time multiplexed, while pipelined accesses are buffered.

ReachingWrites returns all writes in each set which may be visible to any read in the given sets of reads. Visibility is possible if the write may be executed before the read and may have an overlapping address space.

The *BankAndBuffer* function produces a single memory instance from memory reads and writes. Here, each set of accesses is a set of parallel reads or writes to a single port of the memory instance. Accesses in different sets are guaranteed not to occur to the same port at the same time. Therefore, a common banking strategy is found which has no bank conflicts for any set of accesses. This strategy is found using iterative polytope emptiness testing [41].

The required buffer depth d for a pair of accesses a_1 and a_2 to m is computed as

$$d(a_1, a_2) = \begin{cases} 1 & LCA(a_1, a_2) \in Seq \cup Stream \\ dist(a_1, a_2) & LCA(a_1, a_2) \in Pipe \end{cases}$$

where *dist* is the minimum of the depth of the LCA and the dataflow distance of the two direct children of the LCA which contain a_1 and a_2. *Seq*, *Stream*, and *Pipe* are the set of sequential, streaming, and pipelined controllers, respectively. Buffering addressable memories across streaming accesses is currently unsupported. The depth of a set of reads R and writes W is then

$$Depth(R, W) = max\{d(w, a)\ \forall\ (w, a) \in W \times (W \cup R)\}$$

The port of each access within a buffer is determined from the relative distances between all buffered accesses. Spatial requires that no more than one coarse-grained controller or streaming controller is part of a merged instance. The final output of the greedy search is a set of required physical memory instances for memory m.

Acknowledgments

The authors thank the anonymous reviewers for their feedback. This material is based on research sponsored in part by DARPA under agreements number FA8750-17-2-0095, FA8750-12-2-0335, and FA8750-14-2-0240, and NSF grants SHF-1563078 and IIS-1247701. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, NSF, or the U.S. Government.

References

[1] 2015. MyHDL. http://www.myhdl.org/.

[2] 2015. Vivado design suite 2015.1 user guide.

[3] 2016. Vivado High-Level Synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[4] 2017. EC2 F1 Instances with FPGAs âĂŞ Now Generally Available. aws.amazon.com/blogs/aws/ec2-f1-instances-with-fpgas-now-generally-available/.

[5] 2017. Intel FPGA SDK for OpenCL. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[6] 2017. Neon 2.0: Optimized for Intel Architectures. https://www.intelnervana.com/neon-2-0-optimized-for-intel-architectures/.

[7] 2017. Wave Computing Launches Machine Learning Appliance. https://www.top500.org/news/wave-computing-launches-machine-learning-appliance/.

[8] Arvind. 2003. Bluespec: A Language for Hardware Design, Simulation, Synthesis and Verification. Invited Talk. In *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '03)*. IEEE Computer Society, Washington, DC, USA, 249–. http://dl.acm.org/citation.cfm?id=823453.823860

[9] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. 1212–1221.

[10] David Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses. *Queue* 11, 2, Article 40 (Feb. 2013), 13 pages. https://doi.org/10.1145/2436696.2443836

[11] Bruno Bodin, Luigi Nardi, M. Zeeshan Zia, Harry Wagstaff, Govind Sreekar Shenoy, Murali Emani, John Mawer, Christos Kotselidis, Andy Nisbet, Mikel Lujan, Björn Franke, Paul H.J. Kelly, and Michael O'Boyle. 2016. Integrating Algorithmic Parameters into Benchmarking and Design Space Exploration in 3D Scene Understanding. In *PACT*.

[12] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*. ACM, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[13] C. Cascaval, S. Chatterjee, H. Franke, K. J. Gildea, and P. Pattnaik. 2010. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development* 54, 5 (Sept 2010), 5:1–5:10.

[14] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. 2016. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*. IEEE, 327–338.

[15] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. 2013. *Coarse-Grained Reconfigurable Array Architectures*. Springer New York, New York, NY, 553–592. https://doi.org/10.1007/978-1-4614-6859-2_18

[16] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. ACM, New York, NY, USA, Article 83. https://doi.org/10.1145/1188455.1188543

[17] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. 2012. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro* 32, 5 (Sept 2012), 38–51. https://doi.org/10.1109/MM.2012.51

[18] Prabhat K. Gupta. 2015. Xeon+FPGA Platform for the Data Center. http://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf.

[19] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.* 33, 4 (2014), 144–1.

[20] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 85.

[21] Intel. 2015. Advanced NAND Flash Memory Single-Chip Storage Solution. www.altera.com/b/nand-flash-memory-controller.html?_ga=2.108749825.2041564619.1502344247-21903935.1501673108.

[22] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *International Symposium in Computer Architecture (ISCA)*.

[23] Yanqiang Liu, Yao Li, Weilun Xiong, Meng Lai, Cheng Chen, Zhengwei Qi, and Haibing Guan. 2017. Scala Based FPGA Design Flow. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 286–286.

[24] Maxeler Technologies. 2011. MaxCompiler white paper.

[25] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. 2016. Hipa cc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2016), 210–224.

[26] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. 2016. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604.

[27] Luigi Nardi, Bruno Bodin, Sajad Saeedi, Emanuele Vespa, Andrew J. Davison, and Paul H. J. Kelly. 2017. Algorithmic Performance-Accuracy Trade-off in 3D Vision Applications Using HyperMapper. In *iWAPT-IPDPS*. http://arxiv.org/abs/1702.00505

[28] Luigi Nardi, Bruno Bodin, M Zeeshan Zia, John Mawer, Andy Nisbet, Paul HJ Kelly, Andrew J Davison, Mikel Luján, Michael FP O'Boyle, Graham Riley, et al. 2015. Introducing SLAMBench, a Performance and Accuracy Benchmarking Methodology for SLAM. In *ICRA*.

[29] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. 2014. SDA: Software-Defined Accelerator for LargeScale DNN Systems *(Hot Chips 26)*.

[30] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 142–153. https://doi.org/10.1145/2485922.2485935

[31] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 27–40.

[32] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matthew Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. 389–402. https://doi.org/10.1145/3079856.3080256

[33] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2016. Programming Heterogeneous Systems from an Image Processing DSL. *CoRR* abs/1610.09405 (2016). arXiv:1610.09405 http://arxiv.org/abs/1610.09405

[34] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[36] Sajad Saeedi, Luigi Nardi, Edward Johns, Bruno Bodin, Paul Kelly, and Andrew Davison. 2017. Application-oriented Design Space Exploration for SLAM Algorithms. In *ICRA*.

[37] Ofer Shacham. 2011. *Chip multiprocessor generator: automatic generation of custom and heterogeneous compute platforms*. Stanford University.

[38] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 97–108.

[39] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. In *TECS'14: ACM Transactions on Embedded Computing Systems*.

[40] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. 2017. ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 13–26. https://doi.org/10.1145/3079856.3080244

[41] Yuxin Wang, Peng Li, and Jason Cong. 2014. Theory and Algorithm for Generalized Memory Partitioning in High-level Synthesis. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA '14)*. ACM, New York, NY, USA, 199–208. https://doi.org/10.1145/2554688.2554780

[42] Xilinx. 2014. The Xilinx SDAccel Development Environment. https://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf.

[43] Xilinx. 2017. HLS Pragmas. https://www.xilinx.com/html_docs/xilinx2017_2/sdaccel_doc/topics/pragmas/concept-Intro_to_HLS_pragmas.html.

[44] Xilinx. 2017. SDAccel DATAFLOW pragma. https://www.xilinx.com/html_docs/xilinx2017_2/sdaccel_doc/topics/pragmas/ref-pragma_HLS_dataflow.html.

[45] Xilinx. 2017. SDAccel Example Repository. https://github.com/Xilinx/SDAccel_Examples.